



**INSTITUTO POLITÉCNICO NACIONAL**  
**ESCUELA SUPERIOR DE CÓMPUTO**  
**Ingeniería en Sistemas Computacionales**



## **Compiladores**

### **Práctica 4 | Analizador Léxico**

Flores Alvarado Diego Armando – Boleta: 2015340065

Profesor | Saucedo Delgado Rafael Norman

3CV6

Ciudad de México, jueves, 12 de noviembre de 2020

## Introducción

### Descripción del Proyecto

En esta práctica se desarrollará un analizador léxico para lenguaje C en lenguaje LEX ¿Porqué para el lenguaje C? Porque debido a cuestiones de tiempo por el trabajo y mis otras materias se me hace un poco menos compleja la realización de este proyecto para la materia de compiladores.

### ¿Qué es FLEX?

Flex es un generador de análisis léxico, su función es ser una herramienta que genere programas que puedan identificar patrones (por medio de expresiones regulares) en texto. Una vez que recibe un archivo con reglas y expresiones regulares, Flex genera una salida en lenguaje C en un archivo llamado *lex.yy.c*, este archivo puede ser compilado para ser ejecutado y así analizar el lenguaje seleccionado.

Para esta práctica se estará utilizando el sistema operativo de Ubuntu Y Flex en su versión 2.6.4.

## Desarrollo

### 1. Ejemplificación del lenguaje

Para esta práctica, como se mencionó en la introducción, se va a realizar un analizador léxico para el lenguaje C. Se obtuvo el código en LEX para el lenguaje C y fue modificado para obtener las siguientes características.

### 2. Clases Léxicas y Expresiones regulares

Para comenzar la descripción del archivo LEX, se escribieron primero las siguientes definiciones como se puede ver en la Tabla 1:

Tabla 1 - Definiciones LEX

Identificador	Expresión regular
D	[0-9]
L	[a-zA-Z_]
H	[a-zA-F0-9]
E	[Ee][+-]?{D}+
FS	(f F I L)
IS	(u U I L)*

En la Tabla 2 se pueden ver las clases léxicas, así como sus expresiones regulares:

Tabla 2 - Clases Léxicas y ER.

Clase Léxica	Expresión Regular
<b>Palabra Reservada</b>	"break"
	"case"
	"continue"
	"default"
	"double"
	"enum"
	"extern"
	"goto"
	"register"
	"signed"
	"sizeof"
	"static"
	"struct"
	"switch"
	"typedef"
	"union"
	"unsigned"
	"volatile"
<b>Tipo de Dato</b>	"char"
	"const"
	"float"

	"int"
	"long"
	"short"
<b>Ciclo</b>	"for"
	"do"
	"while"
<b>Condicional</b>	"if"
	"else"
<b>Retorno</b>	"return"
<b>Vacío</b>	"void"
<b>Identificador</b>	{L}({L}){D}*
<b>Literal entera base hexadecimal</b>	0[xX]{H}+{IS}?
<b>Literal entera base octal</b>	0{D}+{IS}?
<b>Literal entera base decimal</b>	{D}+{IS}?
<b>Literal carácter</b>	L?'(\. \.[^\'])+'
<b>Literal numérica exponencial</b>	{D}+{E}{FS}?
<b>Literal numérica real exponencial</b>	{D}*".{D}+({E})?{FS}?
	{D}+".{D}*({E})?{FS}?
<b>Literal cadena</b>	L?"(\. \.[^\"])*\"
<b>Delimitador</b>	"..."
	"."
	"{" "<%"
	"}" "%>"
	"."
	"."
	"."
	"[" "<:"
	" ":>"
	"."
	"."
	"."
	"."
<b>Operador compuesto de asignación</b>	">="
	"<="
	"+="
	"-=
	"*="
	"/="
	"%=
	"&="
	"^="
	" ="
<b>Operador a nivel de bits</b>	">"
	"<"
	"&"
	"~"
	"^"

	" "
<b>Operador aritmético</b>	"++"
	"_ _"
	"+"
	"_"
	"**"
	"/"
	"%"
<b>Operador postfijo</b>	"->"
<b>Operador Booleano</b>	"&&"
	"  "
	"!"
<b>Operador de comparación</b>	"<="
	">="
	"=="
	"!="
	"<"
	">"
<b>Operador de asignación</b>	"="
<b>Operador condicional</b>	"?"
<b>Saltos de línea</b>	[ \t\v\n\f]
<b>Resto de caracteres</b>	.
<b>Comentario</b>	"/*"

### 3. Código realizado

Del análisis del código y de las tablas de la sección anterior, se realizó el siguiente código en LEX (que también se adjunta en los anexos en la carpeta de *Código*):

```

1. D      [0-9]
2. L      [a-zA-Z_]
3. H      [a-zA-F0-9]
4. E      [Ee][+-]?{D}+
5. FS     (f|F|l|L)
6. IS     (u|U|l|L)*
7.
8. %{
9. #include <stdio.h>
10. %}
11.
12. %%
13. "/"*   { printf("<Comentario>\n"); }
14.
15. "break" { printf("<Palabra reservada>\n"); }
16. "case"  { printf("<Palabra reservada>\n"); }
17. "char"  { printf("<Tipo de dato>\n"); }
18. "const" { printf("<Tipo de dato>\n"); }
19. "continue" { printf("<Palabra reservada>\n"); }
20. "default" { printf("<Palabra reservada>\n"); }
21. "do"     { printf("<Ciclo>\n"); }
22. "double" { printf("<Palabra reservada>\n"); }

```

```

23. "else"           { printf("<Condicional>\n"); }
24. "enum"           { printf("<Palabra reservada>\n"); }
25. "extern"          { printf("<Palabra reservada>\n"); }
26. "float"           { printf("<Tipo de dato>\n"); }
27. "for"             { printf("<Ciclo>\n"); }
28. "goto"            { printf("<Palabra reservada>\n"); }
29. "if"              { printf("<Condicional>\n"); }
30. "int"             { printf("<Tipo de dato>\n"); }
31. "long"            { printf("<Tipo de dato>\n"); }
32. "register"         { printf("<Palabra reservada>\n"); }
33. "return"          { printf("<Retorno>\n"); }
34. "short"           { printf("<Tipo de dato>\n"); }
35. "signed"          { printf("<Palabra reservada>\n"); }
36. "sizeof"          { printf("<Palabra reservada>\n"); }
37. "static"          { printf("<Palabra reservada>\n"); }
38. "struct"          { printf("<Palabra reservada>\n"); }
39. "switch"          { printf("<Palabra reservada>\n"); }
40. "typedef"          { printf("<Palabra reservada>\n"); }
41. "union"           { printf("<Palabra reservada>\n"); }
42. "unsigned"         { printf("<Palabra reservada>\n"); }
43. "void"            { printf("<Vacio>\n"); }
44. "volatile"         { printf("<Palabra reservada>\n"); }
45. "while"           { printf("<Ciclo>\n"); }
46.
47. {L}({L}|{D})*     { printf("<Identificador>\n"); }
48.
49. 0[xX]{H}+{IS}?    { printf("<Literal entera base hexadecimal>\n"); }
50. 0{D}+{IS}?        { printf("<Literal entera base octal>\n"); }
51. {D}+{IS}?         { printf("<Literal entera base decimal>\n"); }
52. L?'(\\.|[^\\"'])+' { printf("<Literal caracter>\n"); }
53.
54. {D}+{E}{FS}?       { printf("<Literal numerica exponencial>\n"); }
55. {D}*"."{D}+({E})?{FS}? { printf("<Literal numerica real exponencial>\n"); }
56. {D}+"."{D}*({E})?{FS}? { printf("<Literal numerica real exponencial>\n"); }
57.
58. L?"(\\.|[^\\""])*" { printf("<Literal cadena>\n"); }
59.
60. "...              { printf("<Delimitador>\n"); }
61. ">>="             { printf("<Operador compuesto de asignacion>\n"); }
62. "<<="             { printf("<Operador compuesto de asignacion>\n"); }
63. "+="              { printf("<Operador compuesto de asignacion>\n"); }
64. "-="              { printf("<Operador compuesto de asignacion>\n"); }
65. "*="              { printf("<Operador compuesto de asignacion>\n"); }
66. "/="              { printf("<Operador compuesto de asignacion>\n"); }
67. "%="              { printf("<Operador compuesto de asignacion>\n"); }
68. "&="              { printf("<Operador compuesto de asignacion>\n"); }
69. "^="              { printf("<Operador compuesto de asignacion>\n"); }
70. "|="              { printf("<Operador compuesto de asignacion>\n"); }
71. ">>"              { printf("<Operador a nivel de bits>\n"); }
72. "<<"              { printf("<Operador a nivel de bits>\n"); }
73. "++"              { printf("<Operador aritmetico>\n"); }
74. "--"              { printf("<Operador aritmetico>\n"); }
75. "->"              { printf("<Operador postfijo>\n"); }
76. "&&"              { printf("<Operador booleano>\n"); }
77. "||"              { printf("<Operador booleano>\n"); }
78. "<="              { printf("<Operador de comparacion>\n"); }
79. ">="              { printf("<Operador de comparacion>\n"); }
80. "=="              { printf("<Operador de comparacion>\n"); }
81. "!="              { printf("<Operador de comparacion>\n"); }
82. ";"               { printf("<Delimitador>\n"); }
83. ("{"|"<%")        { printf("<Delimitador>\n"); }

```

```

84. ("}"|"%>")      { printf("<Delimitador>\n"); }
85. ","             { printf("<Delimitador>\n"); }
86. ":"             { printf("<Delimitador>\n"); }
87. "="             { printf("<Operador de asignacion>\n"); }
88. "("             { printf("<Delimitador>\n"); }
89. ")"             { printf("<Delimitador>\n"); }
90. ("["|"<:")      { printf("<Delimitador>\n"); }
91. ("]"|":>")      { printf("<Delimitador>\n"); }
92. "."             { printf("<Delimitador>\n"); }
93. "&"             { printf("<Operador a nivel de bits>\n"); }
94. "!"             { printf("<Operador booleano>\n"); }
95. "~"             { printf("<Operador a nivel de bits>\n"); }
96. "-"             { printf("<Operador aritmetico>\n"); }
97. "+"             { printf("<Operador aritmetico>\n"); }
98. "*"             { printf("<Operador aritmetico>\n"); }
99. "/"             { printf("<Operador aritmetico>\n"); }
100. "%"            { printf("<Operador aritmetico>\n"); }
101. "<"            { printf("<Operador de comparacion>\n"); }
102. ">"            { printf("<Operador de comparacion>\n"); }
103. "^"            { printf("<Operador a nivel de bits>\n"); }
104. "|"            { printf("<Operador a nivel de bits>\n"); }
105. "?"            { printf("<Operador Condicional>\n"); }
106.
107. [ \t\v\n\f]    { printf("<Saltos de linea>\n"); }
108. .              { /* ignore bad characters */ }

```

#### 4. Pruebas

Finalmente, corriendo el programa se muestran las siguientes salidas de ejemplo:

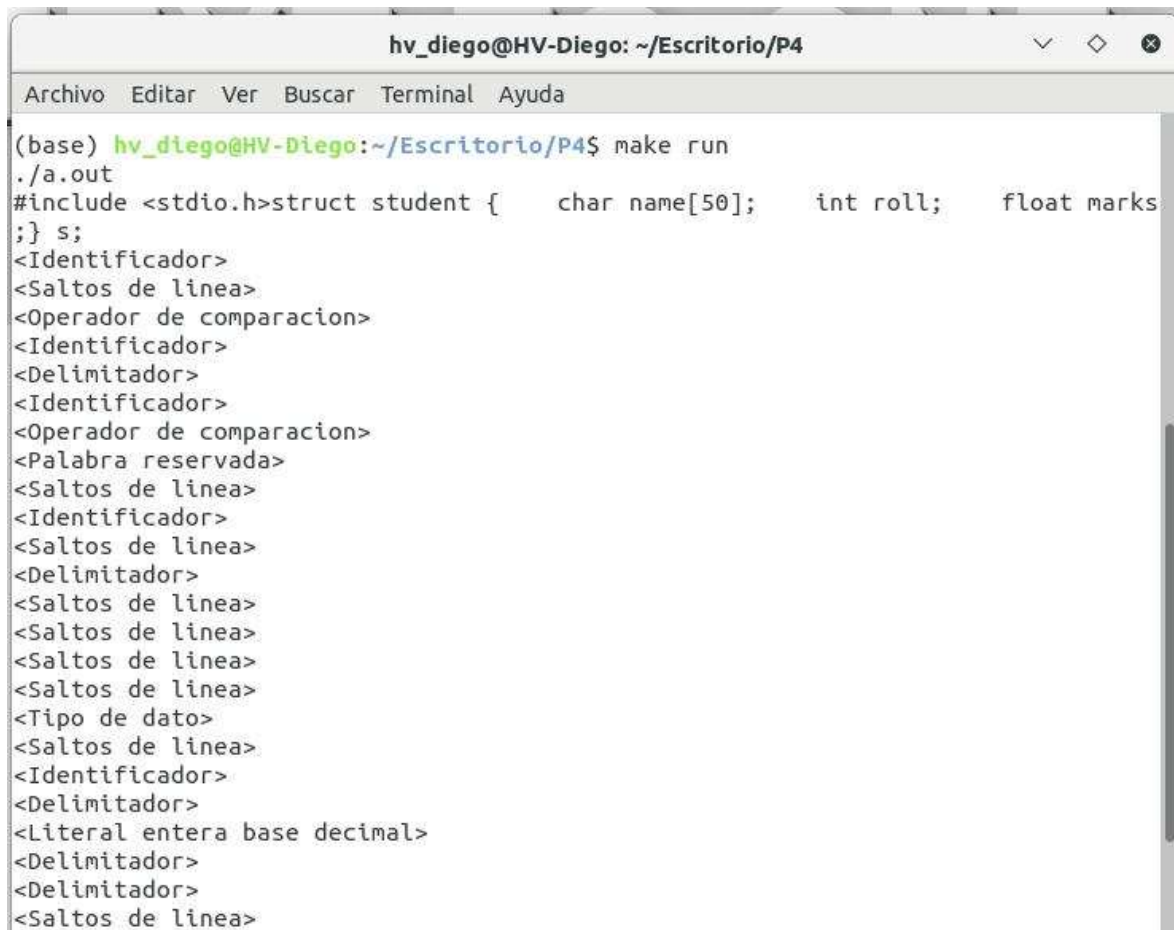


```

hv_diego@HV-Diego: ~/Escritorio/P4
Archivo Editar Ver Buscar Terminal Ayuda
(base) hv_diego@HV-Diego:~/Escritorio/P4$ make run
./a.out
printf("hola, mundo");
<Identificador>
<Delimitador>
<Literal cadena>
<Delimitador>
<Delimitador>
<Saltos de linea>

```

Ilustración 1 - Ejemplo hola mundo



The image shows a terminal window titled "hv\_diego@HV-Diego: ~/Escritorio/P4". The window has a menu bar with "Archivo", "Editar", "Ver", "Buscar", "Terminal", and "Ayuda". The terminal content shows a user running a program with the command `make run`. The program's output is the tokenization of a C code snippet. The code snippet is: `./a.out  
#include <stdio.h>struct student { char name[50]; int roll; float marks  
;} s;`. The tokens are listed line by line: `<Identificador>`, `<Saltos de linea>`, `<Operador de comparacion>`, `<Identificador>`, `<Delimitador>`, `<Identificador>`, `<Operador de comparacion>`, `<Palabra reservada>`, `<Saltos de linea>`, `<Identificador>`, `<Saltos de linea>`, `<Delimitador>`, `<Saltos de linea>`, `<Saltos de linea>`, `<Saltos de linea>`, `<Saltos de linea>`, `<Tipo de dato>`, `<Saltos de linea>`, `<Identificador>`, `<Delimitador>`, `<Literal entera base decimal>`, `<Delimitador>`, `<Delimitador>`, and `<Saltos de linea>`.

```
(base) hv_diego@HV-Diego:~/Escritorio/P4$ make run
./a.out
#include <stdio.h>struct student {    char name[50];    int roll;    float marks
;} s;
<Identificador>
<Saltos de linea>
<Operador de comparacion>
<Identificador>
<Delimitador>
<Identificador>
<Operador de comparacion>
<Palabra reservada>
<Saltos de linea>
<Identificador>
<Saltos de linea>
<Delimitador>
<Saltos de linea>
<Saltos de linea>
<Saltos de linea>
<Saltos de linea>
<Tipo de dato>
<Saltos de linea>
<Identificador>
<Delimitador>
<Literal entera base decimal>
<Delimitador>
<Delimitador>
<Saltos de linea>
```

*Ilustración 2 - Ejemplo con estructura*



## Conclusiones

En esta práctica repasé un poco de términos que no recordaba del lenguaje C, también me ayuda a observar cómo es que un compilador genera los tokens de unas simples líneas de texto. Me sorprende lo rápido que puede llegar a ser, y sólo es una parte de un compilador.

Sinceramente todavía me falta repasar muchas cosas y términos de C que en los siguientes incrementos de esta práctica se irán agregando para que se obtengan mejores resultados en el etiquetado.

## Referencias

- [1] J. Degener, «Lysator,» 1995. [En línea]. Available: <https://www.lysator.liu.se/c/ANSI-C-grammar-l.html>. [Último acceso: 11 Noviembre 2020].
- [2] F. Berzal, «elvex,» [En línea]. Available: <https://elvex.ugr.es/decsai/c/apuntes/tokens.pdf>. [Último acceso: 12 Noviembre 2020].